School of Science, Computing and Engineering Technologies

Object-Oriented Programming

Week 2: Basic Objects with Counter and Shape classes

Overview

In this week, there are **two assessable tasks** 2.1 and 2.2. **Each task contributes 2%** to your final grade. Noting that you need to complete these tasks before coming to your allocated lab. In the lab, there will be verification tasks and short interview to verify your understanding.

Purposes Task 2.1	Practice with properties and use object-oriented encapsulation.
(pages 2-6)	Implement a program that creates and uses a number of counters to explore how objects work. **The task contains personalized requirements.**
Task 2.2 (pages 7-8)	Learn to apply object-oriented programming techniques related to the concept of abstraction. Create a program that can draw rectangular shapes to the screen. **The task contains personalized requirements**

Note: If you are unable to complete theses tasks, you will struggle in this unit. Your tutor may be able assist you with suggestions on how increase your knowledge. You can also participate in HelpDesk sessions

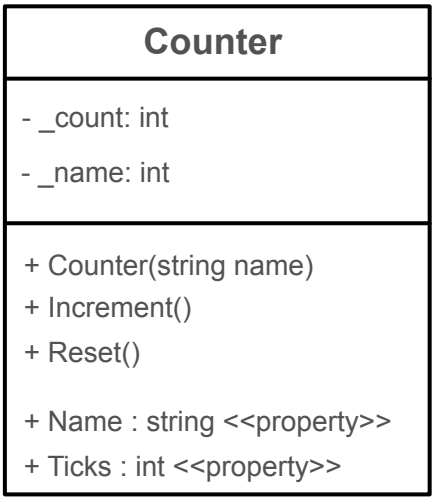
Task 2.1. Instructions

In this task you will create a *Counter* class and explore how fields can be used by an object to store and maintain information.

Each *Counter* object:

- nows its *count* - by using a ***_count*** field to store an integer value,
- nows its *name* - by using a ***_name*** field to store a string value,
- Can generated by using the *constructor* with a string parameter ***name*** that initializes the object's ***_count*** field to zero and sets the object's ***_name*** field to the value of ***name***,
- Can increment object's ***_count*** field by one using the ***Increment*** method,
- Can reset itself by using the ***Reset*** method that sets the ***_count*** field to ,
- Can give you its name via the ***Name*** property (i.e., get …),
- Can change its name via the ***Name*** property by assigning it a new value (i.e., set …),
- Can give you its value via the ***Ticks*** property (i.e., get …).

The following UML class diagram shows the basic outline for this class.



Note: The << … >> annotations in UML are known as stereotypes. They are used to add notes to aspects of the diagram. In this case, <<property>> notes that the ***Name*** attribute here is a property. Properties are *virtual fields*, that is, properties may or may not mapped to actual instance variables. Properties can be read-only, write-only, or read-write. In class *Counter*, property ***Name*** is read-write, whereas property ***Ticks*** is read-only.

1. Create a new *Console App* and name it **CounterTask**.
2. Create a new *Counter* class.
3. Add the private ***_count*** and ***_name*** fields, enabling a *Counter* object to *know* its count and name values.
4. Change the constructor so that it takes a string parameter that is used to set the ***_name*** field of the *Counter* object, and assign to the ***_count*** field.

```
public class Counter
{
    private int _count;
    private string _name;

    public Counter(string name)
    {
        _name = name;
        _count = 0;
    }
}
```

. Add the ***Increment*** method that increases the value of the ***_count*** field by one.

. Add a ***Reset*** method that assigns to the ***_count*** field.

You have now created the code needed to work with *Counter* objects. Each *Counter* object knows its *count* and *name* and can increment and reset its count value. Notice, the things a *Counter* object knows are *hidden* within the object (due to the ***private*** modifier on the fields). This is one of the guiding principles of *object-oriented encapsulation*. Object-oriented encapsulation is a mechanism that allows you to hide specific information and control access to the object s internal state. In general, you achieve object-oriented encapsulation by making all instance variables ***private***, and provide read or write access to instance variables via ***public*** methods only.

The keywords ***private*** and ***public*** serve as scope modifiers in C#. A feature marked ***private*** is only visible within the scope of the defining class and its objects. Features marked ***public*** are visible to all clients of a class and its objects. The term client refers to other classes, other objects, and even other applications. You should aim at achieving a suitable balance between public and private visibility within a class and its objects. Object-oriented encapsulation is not a mechanism whose principles are set in stone sometimes it can be beneficial to loosen access restrictions if the domain abstraction calls for it.

C# includes a feature, called *properties*, that allows you to provide access to data in a controlled way. From the outside, properties look and feel like instance variables of an object. Hence, they are also known as *virtual fields*. However, properties are actually mapped to a pair of methods: ***get*** to retrieve a value, and ***set*** to update a value. Properties are not simply used to provide access to instance variables. Instead, you often use properties to perform safety checks or compute composite values for a set of attributes.

7. Create a ***Name*** property for *Counter* objects using the following code:

```
public class Counter
{
    private string _name;

    public string Name
    {
        get
        {
            return _name;
        }
        set
        {
            _name = value;
        }
    }
}
```

Properties have the general format as shown below. However, you can add any code you want within the get and set methods, as long as get returns a value and set changes a value.

```
    public [TYPE] PropertyName
    {
        get
        {
            return …
        }
        set
        {
            … = value;
        }
    }
```

8. Create the ***Ticks*** property for the *Counter* class. ***Ticks*** is a read-only property that returns the value of the ***_count*** field.

Hint: Read-only properties have only a ***get*** method, write-only properties have only a ***set*** method.

Your *Counter* class is now complete. Build your solution and fix any error before you proceed.

9. Return to the *Program.cs* file.

10. Implement the following pseudocode for the static method ***PrintCounters*** method:

```
    PrintCounters(counters)
1:  // parameter counters is an array of Counter objects
2:  foreach c in counters
3:      Tell Console to WriteLine with the format "{0} is {1}"
4:           and the result of Tell c to Name
5:           and the result of Tell c to Ticks
```

Tips:

To declare a static method, you need to annotate the signature of the method with the keyword ***static***. For example, ***static public void Print*** (string names) …

Foreach loops are a simple way of traversing over all of the elements of an array in C#. For example `foreach (string name in names) { … }`

The loop variable c in the pseudocode is a *Counter* object. In C#, you need to write *Counter* c so that the loop variable has a proper type.

Note: *Console*'s ***WriteLine*** method can take a variable number of parameters. The marker means inject the 1st value following the string at this point. For example:

```
Console.WriteLine("Hello, {0}{1}", "World", "!");
```

Please note, both ***Main*** and ***PrintCounters*** are static methods. Static methods are not associated with any object. Hence, you do not need to create a *Program* object to use ***PrintCounters***.

```
internal class Program
{
    private static void PrintCounters(Counter[] counters)
    { … }

    static void Main(string[] args)
    { … }
}
```

11. Use the following pseudocode to implement the **Main** method.

```
     Main()
 1:  Let myCounters be an array of three Counter objects
 2:  myCounters[0] := new Counter with name "Counter 1"
 3:  myCounters[1] := new Counter with name "Counter 2"
 4:  myCounters[2] := myCounter[0]
 5:  for i := 1 to 9
 6:       Tell myCounters[0] to Increment
 7:  for i := 1 to 14
 8:       Tell myCounters[1] to Increment
 9:  Tell Program to PrintCounters(myCounters)
10:  Tell myCounters[2] to Reset
11:  Tell Program to PrintCounters(myCounters)
```

In pseudocode, a **for** i: **to** 3 statement has four iterations [0,1,2,3]. How many iterations does **for** i: 1 **to** 4 have?

Hint: You can declare the array using

```
Counter[] myCounters = new Counter[3];
```

12. Create a new Reset method, named, ResetByDefault, in the Counter class to reset the corresponding count to the integer valued at 21474836XXXX, where the XXXX is the last four digits of your student ID. How can you implement this step?

13. Tell the Counter to increase the count value by 5. Does the code still run without any bugs/crash? What is the reason behind? You can provide the answers in the comments in your code.

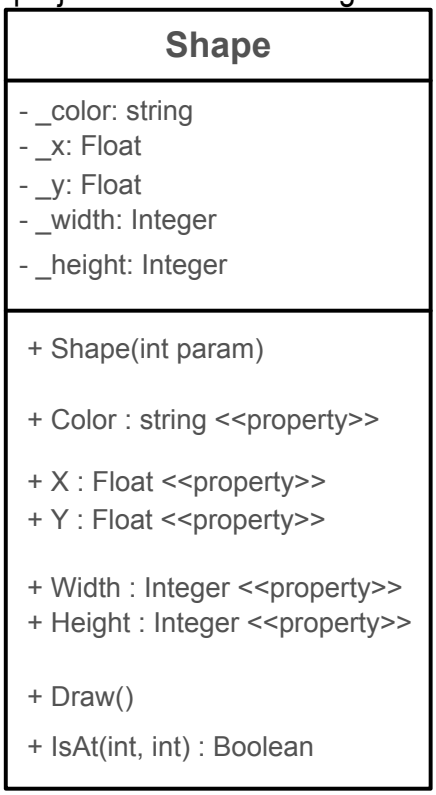
Hint. Please read Microsoft Learn portal
https learn.microsoft.com en-us dotnet csharp language-reference statements checked-and-unchecked

14. Compile and run your program.
15. Save and back up your programming project and solution. Show to your tutor when you arrive the lab.

Task 2.2 Instructions

Over the course of the next few weeks you will develop a simple shape drawing program with Graphic User Interface. In this week, you begin by creating a *Shape* class. You should create this project separated from the previous task 2.1. In this way, we can easily extend the project in the following weeks.

1. Create a new ShapeDrawing project
2. Add a *Shape* class to your project. Use the following UML class diagram as a guide.



3. . The type *Color* is defined as a *string* at this stage. In Week 4, we will replace the datatype using SplashKit.

```
public class Shape
{
    private string _color; …
```

4. In the constructor, initialize ***_color*** to a string *"Color. Azure"* if the first letter of your first name is from A to L. Otherwise, set by *"Color.Chocolate"*. In that constructor, set ***_x*** and ***_y*** to 0.0f (the suffix f makes . a float value), and both the _width and _height are assigned to the value of a given parameter called param. When you later create objects based on this Shape class, please specify the parameter to be 1 , where is the last two digits of your student ID.

5. The ***Draw*** method will print out the essential information including the shape's color, position, and dimension. In Week 4, we will replace this method with a drawing function from SplashKit

```
public class Shape
{
    …
    public void Draw()
    {
       Console.WriteLine("Color is " + _color);
       Console.WriteLine("Position X is " + _x);
          ....
    }
```

6. Add the ***IsAt*** method which takes ***two*** integers (int xInput, int yInput) representing a point in 2d space - like a point on the screen), and returns a Boolean to indicate if the shape is at that point. You need to return true if the point *pt* is within the shape's area (as defined by the shape's coordinates).

Tip: What does it mean for a point to be considered inside the area of a rectangle?Assuming, (x1,y1) and (x2,y2) are the top-left and bottom-right corners of a rectangle, respectively. A 2Dpoint (xInput, yInput) is inside the rectangle only if (xInput > x1 && xInput < x2 && yInput > y1 && yInput < y2)

7. Add all properties (as defined in the UML diagram) to class *Shape*.
8. Return to the *Program.cs* file.
9. In ***Main***,
    - Add a ***myShape*** local variable of the type *Shape*.
    - Assign ***myShape***, a **new** *Shape* object using the Shape constructor
    - Tell myShape to Draw itself
    - Compile and run to obtain the result from the console
10. Save and backup your code using Cloud, USB, or any drive

When you arrive at your lab, you will receive the verification tasks.
See you very soon. I